Software-Reengineering
Umgang mit Software-Altlasten

Jürgen Ebert
Institut für Softwaretechnik
Universität Koblenz-Landau
ebert@uni-koblenz.de

Zusammenfassung

Der Umgang mit Altsoftware bringt zahlreiche Herausforderungen. In Sanierungsprojekten sind verschiedene Reengineering-Aufgaben zu lösen. Zahlreiche meist graph-basierte Reverse-Engineering-Techniken zusammen mit bekannten Techniken der Software-Entwicklung sind werden hierfür eingesetzt. Es ist Aufgabe des Teilgebietes Software-Reengineering, die für diese Zwecke erforderlichen Modelle und Techniken zu definieren, um sie als Dienste in repository-zentrierten Werkzeugen zur Lösung von Reengineering-Aufgaben zur Verfügung zu stellen.

1. Problemstellung

Moderne Softwaretechnik kann sich nicht allein mit den Fragen der Entwicklung neuer Softwaresysteme beschäftigen. Der größte Teil der praktisch eingesetzten Software ist bereits mehr als ein Jahrzehn alt und bedarf weiterhin einer intensiven Wartung und Pflege. Es gibt einen riesigen Bestand an *Altsoftware* (legacy software) – Sommerville [S01] schätzt ihn auf über 250 Milliarden Zeilen – in alten Programmiersprachen wie Cobol, Fortran, RPG oder Assembler, der auf konventionelle Host-Systeme zugeschnitten ist, die sich von modernen Systemarchitekturen deutlich unterscheiden.

Altsoftware verursacht eine Menge zusätzlicher Probleme bei der Wartung. Veraltete Programmiersprachen in teilweise schon verschollenen Dialekten werden darin verwendet. Viele dieser Programme sind durch den Programmierstil, durch einen Mangel an Dokumentation und Kommentierung und durch die Verwendung teilweise undurchsichtiger Optimierungen und trickreicher Programmierung nicht mehr verstehbar und daher auch nur sehr schwer wartbar. Hinzu kommt, dass jahrelange korrektive und adaptive Wartung Softwaresysteme in ihrer Struktur zerstört und redundanten Code erzeugt, so dass die noch vorhandene Klarheit und Übersichtlichkeit weiter verloren geht.

Gleichzeitig ist Altsoftware aber auch einer Einbindung in weitere Entwicklungen unterworfen. Die Einbindung existierender Host-Anwendungen in das Web erfordert die Erzeugung neuer Schnittstellen, die Portierung auf lokale Netze bedingt eine Aufteilung der Anwendungen zum Zwecke der Verteilung, und das Verknüpfen mit moderner Middleware fordert die Verkapselung der Altsoftware in Objekte. Hinzu kommen noch äußere Anlässe, die größere Änderungsaktivitäten erfordert, wie es beispielsweise das sog. Jahr-2000-Problem oder die Euro-Umstellung taten.

Ferner ist die Änderung oder gar Überarbeitung von Altsoftware besonders dadurch problematisch, dass sie im allgemeinen eng mit den Geschäftsprozessen verknüpft ist, was einer isolierten Änderung der Software ohne Berücksichtigung der Abläufe entgegensteht. Gleichzeitig enthält Altsoftware häufig viel Unternehmenswissen, das außerhalb des Quellcodes nirgendwo dokumentiert ist. Altsoftware stellt somit also auch einen hohen Wert an sich dar und hat eine zentrale Bedeutung.

Reengineering ist der Oberbegriff für alle Prozesse, deren Ziel die qualitative Verbesserung und Aufbereitung von Software ist. Ähnliche und im Wesentlichen hierzu synonyme Begriffe sind auch: Wiederaufbereitung, Modernisierung, Renovierung oder Redevelopment. Meistens werden Reengineering-Maßnahmen im Rahmen eines Projekts durchgeführt. Dann spricht man auch von *Software-Sanierung*.

Zur Durchführung von Sanierungsmaßnahmen stellen sich je nach Problemsituation verschiedene *Reengineering-Aufgaben*, die unterschiedliche Herausforderungen an die Softwaretechnik darstellen. Typische Reengineering-Augaben sind beispielsweise

- Sprachportierung, etwa von Cobol in eine moderne objekt-orientierte Sprache oder auch bereits innerhalb der objekt-orientierten Welt von Smalltalk nach Java. Diese Aufgabe erfordert eine tiefe Durchdringung der von Syntax, der Semantik und der Pragmatik der beteiligten Sprachen.

- Wrappergenerierung, also die Verkapselung von Altsoftware in Objekten. Diese Aufgabe erfordert die Entdeckung und die vollständige Erfassung der Schnittstellen und die Generierung von passendem Verkapselungs- und Anwendungscode.
- Datenbankmodernisierung, wie etwa der Übergang von einer hierarchischen Datenbank in eine relationale Datenbank. Diese Aufgabe erfordert die Transformation der Datenbeschreibung, die Migration der Daten und ebenfalls eine Anpassung der Anwendungssoftware.

Alle diese komplexen Reengineering-Aufgaben erfordern zahlreiche verschiedene, teilweise sehr anspruchsvolle Methoden und Techniken zur Behandlung von Software, die in unterschiedlichen Kombinationen zur Lösung der Aufgaben eingesetzt werden müssen.

2. Vorgehensmodelle

Ein Reengineering-Vorhaben, das als Sanierungsprojekt durchgeführt wird, besteht i.a. aus einer Reihe von verschiedenen Tätigkeiten. Die Durchführung derartiger Sanierungsprojekte erfordert daher ein überlegtes Vorgehensmodell, wie man es auch bei der Erstellung von Software macht. Typische Aktivitäten eines Sanierungsprojektes sind [EGW96]

- Bestandsanalyse, d.h. eine Inventur der vorhandenen Software,
- Anwendungsverstehen, d.h. ein Grundverständnis des Systems von Seiten der Anwendung,
- Programmverstehen, d.h. eine Identifikation der genauen Abläufe auf Programmebene,
- Reverse Engineering, d.h. die Extraktion von Information aus dem Quellcode,
- Forward Engineering, d.h. die Neuentwicklung von Teilen des Systems in überarbeiteter Form,
- Validierung, d.h. die Sicherstellung, dass das veränderte System die gleiche Funktionalität aufweist,
- Migration, d.h. die Umstellung der Daten und Abläufe auf das erneuerte System.

Die für die Softwaretechnik anspruchsvollste Tätigkeit ist hierbei das *Reverse-Engineering*. Reverse Engineering ist die (Wieder-)Gewinnung höherer Abstraktionsebenen aus dem Quellcode. Hier sind Analyse-Methoden gefragt, die im Stande sind, aus dem Quellcode Informationen zurückzugewinnen, die im Zuge der Entwicklung vorhanden waren.

Byrne [By92] veranschaulich die Bedeutung des Reverse-Engineering, wie in Abbildung 1 dargestellt, als Überführung der Software auf eine höhere Ebene, die es dann erlaubt, abstraktere, d.h. weitreichendere Transformationen durchzuführen, um sie dann als Ausgang einer erneuten Entwicklung verwendet werden können. Hierdurch wird das Reverse Engineering als Folge dreier Schritte Abstraktion-Änderung-Verfeinerung (extract-transform-rebuild) verstehbar.

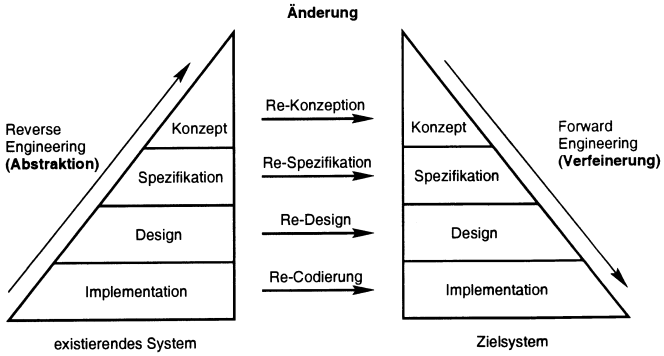


Abb. 1: Software-Reengineering [By92]

Reverse Engineering stellt neue, d.h. von der Programmentwicklung bisher noch nicht bekannte Herausforderungen. Man kann viele *Reverse-Engineering-Aufgaben* identifizieren, für die effiziente Lösungen entwickelt werden müssen, beispielsweise

- Extraktion von Subsystemen
- Herleitung der Architektur

- Erkennung von möglichen Klassen in prozeduraler Software
- Finden von doppeltem oder überflüssigem Kode
- Übersetzung einzelner Bausteine in andere Programmiersprachen
- Bestimmung von Schnittstellen in schwach strukturierten Systemteilen

Es ist Aufgabe der Forschung, diese Fragestellung des Reverse Engineering scharf zu definieren und Lösungen hierzu anzubieten. Alle diese Lösungen sollen einen möglichst großen Anwendungsbereich haben und miteinander kombinierbar sein. Die Kombinierbarkeit erfordert gemeinsame Bezüge, die mit einer Modellierung der Software und der zugehörigen Artefakte auf den höheren Abstraktionsebenen geschaffen werden können.

3. Modelle

Die Abstraktionen, die im Zuge des Reverse Engineering hergeleitet werden, sind i.a. verschiedene diskrete Sichten auf die Software. Je nachdem, welche Aufgabe gelöst werden soll werden unterschiedliche Bestandteile identifiziert und extrahiert und auf ihre Beziehungen zueinander untersucht. Nahezu alle diese Sichten lassen sich als Graphen beschreiben. Lediglich die Bedeutung der Knoten und Kanten ist jeweils unterschiedlich.

Um eine möglichst mächtige, bequeme und adäquate Modellierung zu ermöglichen, sollte man Graphen verwenden, die eine hohe Modellierungsfähigkeit haben. Ein sehr allgemeiner Fall sind dabei *TGraphen* [EW+96], d.h. typisierte, attributierte und angeordnete gerichtete Graphen.

Für eine konkrete Sicht entspricht dann die Definition der zugehörigen Abstraktionen der Definition einer Klasse von TGraphen. Derartige *Graphklassen* können beispielsweise durch erweiterte Entity-Relationsship-Schemata (*EER-Schemata)* spezifiziert werden. Ein EER-Schema erfasst die erlaubten Knoten- und Kantentypen mit ihren möglichen Attributen und deren Zusammenspiel. Im Rahmen des Reverse Engineering werden auf diese Weise diejenigen Graphen beschrieben, die die Informationen so repräsentieren, wie sie für die einzelnen Aufgaben gebraucht werden. Gleichzeit charakterisiert ein solches Schema die Struktur der Daten innerhalb des Repositorys entsprechender Werkzeuge (s.u.).

Abbildung 2 enthält als Beispiel ein kleines Code-Fragment und als den *abstrakten Syntaxgraphen*, der das Code-Fragement repräsentiert. Dieser TGraph ist in Knoten und Kanten damit typisiert, welche Bedeutung der Knoten hat. Einige Knoten enthalten zusätzlich konkrete Attribute, und die Kanten, die aus den Knoten heraus- und die in die Knoten hineingehen, sind so angeordnet, dass aus dem gesamten Graphen der Ausgangstext bis auf Layout-Details wieder hergestellt werden kann.

```
while x > 0 do
   repeat
      y := y + 1;
   until (y = x);
   x := x - 1
end.
```

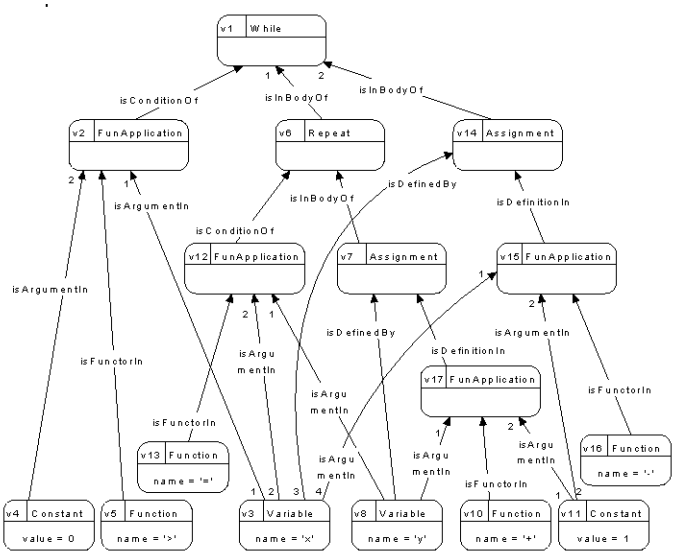


Abb. 2: Programmfragment und Abstrakter Syntaxgraph

Man sieht an diesem Beispiel, dass ein abstrakter Syntaxgraph eine klare, von der Programmiersprache vorgegebene Form hat. Abbildung 3 beschreibt die Klasse der gültigen abstrakten Syntax-Graphen für die verwendete Beispielsprache durch ein entsprechendes Schema (hier in der Notation der UML-Klassendiagramme dargestellt). Dadurch werden die Knotentypen und die Kantentypen, die Attribute und die Kardinalitäten der erlaubten Graphen eindeutig festgelegt. Weitere Kontextbedingungen können durch passende Constraint-Sprachen hinzugefügt werden [EW+96].

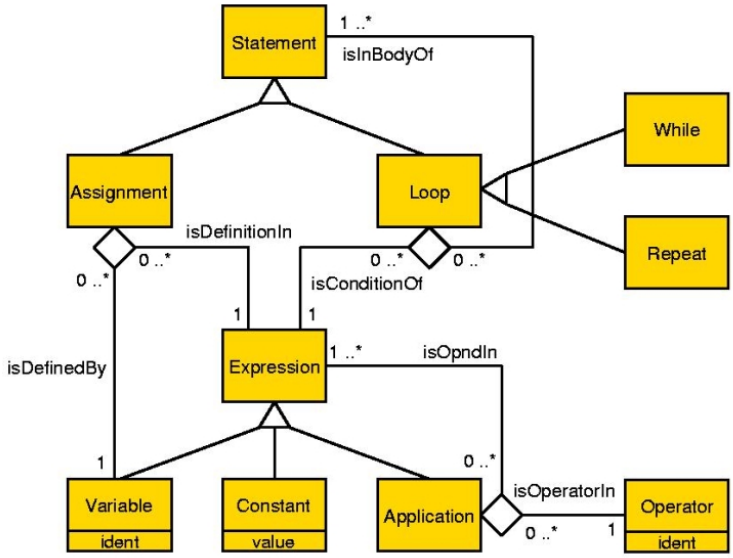


Abb. 3: Graphklasse der Abstrakten Syntaxgraphen

Die Charakterisierung von Graphklassen durch derartige Schemata erlaubt die Modellierung der verschiedenen Graphklassen auf unterschiedlichen Granularitätsstufen, für unterschiedliche Sprachen und aus unterschiedlichen Sichten. Für die verschiedenen Reverse-Engineering-Aufgaben sind i.a. verschiedenartige Graph-Schemata erforderlich, je nachdem welche Information man repräsentieren möchte. Weitere typische Graphklassen sind beispielsweise

- Kontroll-Graphen
- Kontrollfluss-Graphen
- Datenfluss-Graphen
- Programm-Abhängigkeits-Graphen
- Aufruf-Graphen

4. Techniken

Für die genauere Beschreibung der Lösung von Reverse-Engineering-Aufgaben reicht die Einführung der verschiedenen Graphklassen noch nicht aus, sondern es sind auch *Techniken* erforderlich, die es erlauben, die Lösungen der Aufgaben schrittweise zusammen zu bauen. Immer wieder verwendete Basistechniken sind beispielsweise

- Parsing/Unparsing, d.h. die Überführung von Programm in Graphen und umgekehrt,
- Kontrollfluss-Bestimmung, d.h. die Berechnung der möglichen Übergänge bei der Ausführung von Programmen.
- Datenfluss-Berechnung, d.h. die Ermittlung der Beziehungen zwischen Zuweisungen an Variablen und deren Benutzung,
- Pointeranalyse, d.h. die Abschätzung der Objekte, auf die Zeigervariablen referieren können.

Diese Basistechniken sind werden teilweise auch in anderen Teilgebieten der Softwaretechnik eingesetzt. Für den Umgang mit den berechneten Informationen verwendet man oft noch weitere aus anderen Bereichen der Informatik bekannte Techniken zur Strukturierung, wie beispielsweise

- Konzeptanalyse zur Strukturierung von Informationen anhand ihrer Attribute,
- Clusteranalyse zur Gruppierung von Informationen aufgrund ihrer Ähnlichkeit,
- Visualisierung zur Darstellung von ermittelter Information in für den Menschen adäquater Form,

5. Beispiel: Slicing

Am Beispiel des *Program Slicing* [W84] wird im Folgenden für ein einfaches Programm die Verwendung der Modellierung von Abstraktionen durch Graphen und die Verwendung von Basistechniken zur Lösung einer speziellen Reverse-Engineering-Fragestellung demonstriert.

Eine (Rückwärts-)*Slice* eines Programms ist ein (möglichst kleines) Programmfragment, das den Wert einer Variablen *v* in einer Anweisung *s* bereits vollständig bestimmt. Durch das *Slicing-Kriterium* <*s,v*> wird dieses Programmfragment charakterisiert. Das Programm in Abbildung 4(a) beispielsweise sollte, wie sein Name `printSum` ausdrückt, eigentlich nur die Summe der Zahlen von 1 bis n drucken, enthält offensichtlich aber mehr Anweisungen als hierzu erforderlich. Die Berechnung einer Slice zum Kriterium <*14,s*> könnte, wie in Abbildung 4(b) gezeigt, das Programm entsprechend verkleinern.

```
01 printSum (int n)
02 {
03    int i = 1;
04    int s = 0;
05    int p = 1;
06
07    while (i <= n) {
08       s += i;
09       p *= i;
10       i += 1;
11    }
12
13    println (i);
14    println (s);
15    println (p);
16 }
```

(a)

```
01 printSum (int n)
02 {
03    int i = 1;
04    int s = 0;
05
06
07    while (i <= n) {
08       s += i;
09
10       i += 1;
11    }
12
13
14    println (s);
15
16 }
```

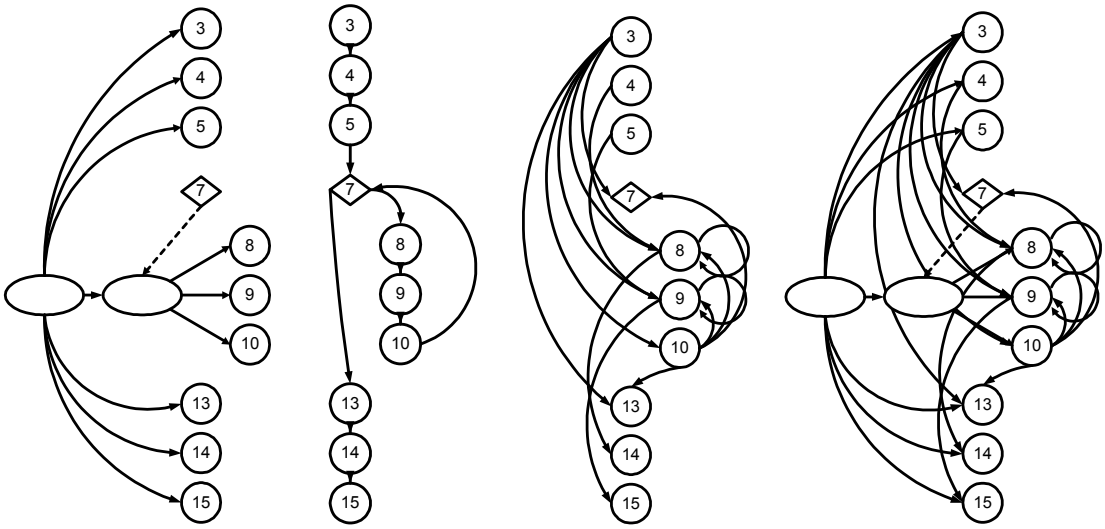
(b)

Abb. 4: Programm und Slice

Es ist aus der Literatur bekannt [FOW87], dass die Berechnung einer Slice in einem einfach strukturierten Programm durch die Berechnung verschiedener Abstraktionen (Graphen) erfolgen kann:

1. Berechnung des *Kontroll-Graphen* G_{CG}, der hier mit dem Struktur-Graphen G_{SG} zusammenfällt,
2. Berechnung des *Kontrollfluss-Graphen* G_{CFG} aus dem Struktur-Graphen,
3. Berechnung der *Datenfluss-Graphen* für die einzelnen Variablen aus dem Kontrollfluss-Graphen und damit die Berechnung des gesamten Datenfluss-Graphen $G_{DFG,}$
4. Berechnung des *Programm-Abhängigkeits-Graphen* $G_{PDG,}$ der die Vereinigung des Kontroll-Graphen G_{CG} mit dem Datenfluss-Graphen G_{DFG} darstellt.

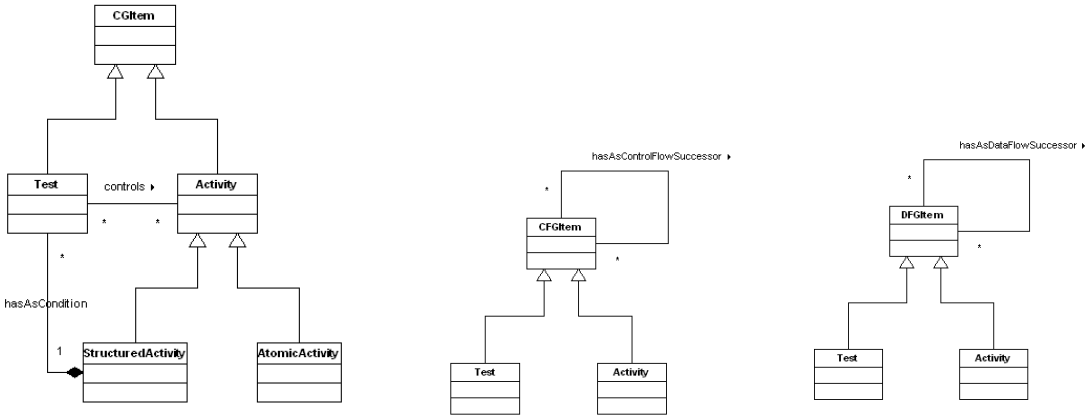
Die Struktur dieser Graphen lässt sich durch (hier sehr einfach gehaltene) Schemata wie in Abbildung 6 modellieren. Durch einfache *Graph-Algorithmen* oder durch entsprechende *Graph-Anfragesprachen* kann man aus dem Syntaxgraphen die entsprechenden Abstraktionen berechnen (Abbildung 5).



G_{SG} G_{CFG} G_{DFG} G_{PDG}

Abb. 5: Graphen

Den Strukturgraphen G_{SG} des vorliegenden strukturierten Programms gemäß Schema SG in Abbildung 6 erhält man durch eine Analyse der syntaktischen Struktur beim Parsing. Durchläuft man diesen Graphen entsprechend der Anordnung der Kanten, so kann man hieraus direkt den Kontrollfluss herleiten. Das Schema CFG legt die zur Beschreibung verwendet Graphenstruktur fest. Den Datenfluss bestimmt man dann am besten für jede Variable *v* einzeln. Vom Knoten i zum Knoten j wird eine Datenflusskante gezogen, wenn der Wert von *v* in *i* gesetzt, in *j* verwendet wird und dazwischen keine Änderung stattfindet. (definition-use-chain). Für diese Berechnung benötigt man den Kontrollfluss. Die Vereinigung der einzelnen Datenfluss-Graphen der verschiedenen Variablen liefert den Datenfluss-Graphen G_{DFG} des gesamten Programms gemäß Schema DFG. Der Kontrollgraph G_{SG} und der Datenflussgraph G_{DFG} zusammen enthalten die gesamte für das Slicing relevante Information. Ihre Vereinigung wird Programm-Abhängigkeits-Graph G_{PDG} genannt.



SG CFG DFG

Abb. 6: Graphklassen

Berechnet man jetzt durch ein Graph-Suchverfahren diejenigen Knoten, von denen aus der Knoten für Statement 14 erreichbar ist, so erhält man den in Abbildung 7 gezeigten Teilgraphen. Die in diesem Graphen markierten Knoten stellen genau die zur Berechnung der Slice erforderlichen Anweisungen dar und führen zu dem in Abbildung 4(b) angegeben Teilprogramm.

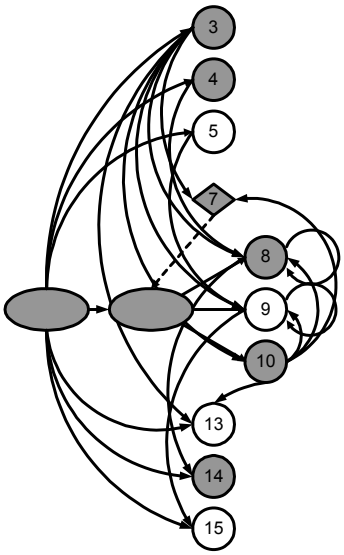


Abb. 7: Ergebnisgraph

6. Werkzeuge

Reengineering-Werkzeuge müssen den in Abbildung 1 beschrieben extract-transform-rebuild-Zyklus unterstützen. Sie sollen i.a. mehrere Basistechniken anbieten und möglichst deren Kombination unterstützen, damit ernsthafte Reengineering-Aufgaben gelöst werden können. Zusätzlich ist eine Integration auch mit Programm-Entwicklungs-Werkzeugen für das Forward Engineering erforderlich.

Es gibt zahlreiche verschiedene derartige Werkzeuge, die ebenfalls sehr verschiedenartige Technologien verwenden. Gemeinsam ist den verschiedenen Werkzeugen allerdings, das sie nahezu alle repository-basiert sind. Wie in Abbildung 7 skizziert arbeiten sie auf einer zentralen Datenhaltung (Repository). Die Spannbreite der angewandten Technologien ist allerdings sehr breit. Sie reicht von Text-Files über relationale Datenbanken und Graphenspeicher bis hin zu Prolog-Klauselmengen und XML-Files.

Ein Beispiel ist die Werkzeugumgebung GUPRO [EK+02], die auf in TGraphen gespeicherten Modellen eine Unterstützung von integriertem Anfragen und Browsing bietet, wodurch Software auch heterogener Sprachen und auf verschiedenen Granularitätsstufen untersucht werden kann. In Kombination mit einzelnen Graphenalgorithmen ist so eine Unterstützung verschiedener Reverse-Engineering-Aufgaben möglich.

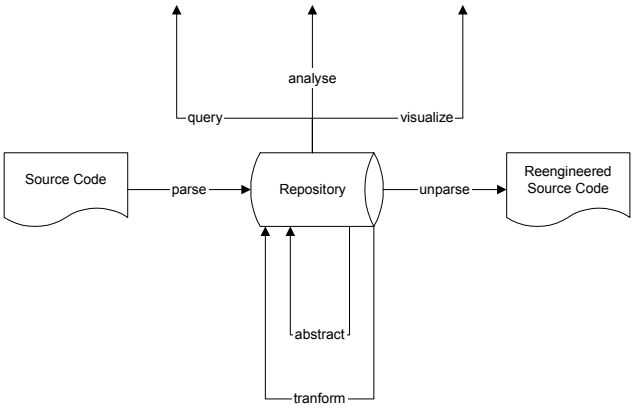


Abb. 8: Struktur von Reengineering-Werkzeugen

Eine Standardisierung der Repository-Technik und der verwendeten Abstraktionen, die durch Schemata beschrieben werden können, ist bisher noch nicht geschehen. Um die *Interoperabilität* von Reengineering-Werkzeugen herzustellen, wurde ein gemeinsames XML-basiertes Austauschformat definiert (GXL [WKR01]). Zur Zeit erfolgen verschiedene Versuche der Festlegung der benötigten Graphklassen für reale Reengineering-Aufgaben. Hier stellt sich eine weitere große Aufgabe für die Reengineering-Forschung, nämlich die Definition gemeinsamer Referenz-Schemata für die Interoperabilität von Reengineering-Werkzeugen.

7. Zusammenfassung

In diesem Überblickspapier wurde beschrieben, wie sich aus den Reengineering-Aufgaben im Umgang mit Altsoftware konkrete Fragestellungen der Entwicklung von Modellen, Techniken und Werkzeugen ergeben, die neue Herausforderungen an die Wissenschaft darstellen.

Es ist Aufgabe des Teilgebiets Reengineering der Softwaretechnik die wesentlichen Abstraktionsmodelle für das Reverse Engineering zu definieren und einzelne Basistechniken zu entwickeln, die zur Berechnung dieser Modelle verwendet werden können, und diese so zu präzisieren, dass sie als Basisdienste in Werkzeugumgebungen eingestellt werden können.

Eine graph-basierte Sicht erlaubt eine klare Beschreibung der Probleme und eine Definition der Verfahren durch Graphenalgorithmen. Die Erstellung eines Katalogs relevanter Referenzmodelle zusammen mit zugehörigen Berechnungstechniken stellt das Forschungsprogramm des Fachgebiets für die nächsten Jahre dar.

Literatur:

[By92] Byrne, E.J.: A Conceptual Foundation for Software Re-engineering. in: Proceedings of the International Conference on Software Maintenance and Reengineering (ICSM '92). IEEE Computer Society, 1992, S. 226-235

[EGW96] Ebert, Jürgen; Gimnich, Rainer; Winter, Andreas: Wartungsunterstützung in heterogenen Sprachumgebungen, Ein Überblick zum Projekt GUPRO. In F. Lehner (Editor), Softwarewartung und Reengineering - Erfahrungen und Entwicklungen, Gabler, Wiesbaden, 1996, S. 263-275.

[EK+02] Ebert, Jürgen; Kullbach, Bernt, Riediger, Volker; Winter, Andreas: GUPRO – Generic Understanding of Programs - An Overview. Electronic Notes in Theoretical Computer Science 72 No. 2 (2002).
s.a. `http://www.gupro.de`

[EW+96] Ebert, Jürgen; Winter, Andreas; Dahm, Peter; Franzke, Angelika; Süttenbach, Roger Graph Based Modeling and Implementation with EER/GRAL. In: Bernhard Thalheim (Editor) Conceptual Modeling - ER'96. Springer, Berlin 1996, LNCS 1157, 163-178

[FOW87] Ferrant, Jeanne; Ottenstein; Karl J.; Warren, Joe D. The Program Dependence Graph and Its Use in Optimization. ACM TOPLAS 9(1987,3), 319-349

[S01] Sommerville, Ian: Software Engineering. 6th Edition. Addison-Wesley, Harlow England, 2001

[WKR01]Winter, Andreas; Kullbach, Bernt; Riediger, Volker: An Overview of the GXL Graph Exchange Language. In: S. Diehl (Editor) Software Visualization, International Seminar Dagstuhl Castle, Springer, Berlin, 2001.
s.a. `http://www.gupro.de/GXL`.

[W84] Weiser, Mark: Program Slicing. IEEE Transactions on Software Engineering 10(1984,4),352-357